实践指导

吴清晏 (61522314)
东南大学吴健雄学院

前言

感谢您使用《openEuler 内核开发实践指导》!

本手册是由董恺老师发起，学生孙彦林基于《中国科学院大学 < 操作系统 > 课程实验课指导书》进行编写的课程实践指导手册。为了满足实验的最新需求和学生的学习需求，我们对原有内容进行更新与修复，通过分析和解答学生在实验中遇到的问题，以确保内容的准确性和完整性。

本手册旨在引导您进行《openEuler 内核开发实践》课程实践实验，提供全面而系统的指导。通过本实验，您将有机会亲身参与开源操作系统的开发，学习并应用现代操作系统的设计原则、调度算法、内存管理和设备驱动等关键概念。

我们特别强调实验的流程性描述，以确保您能够按照正确的顺序进行实验操作，并理解每个步骤的目的和意义。每个实验步骤都经过精心设计和验证，旨在帮助您逐步掌握内核开发所需的技能和知识。

董恺老师是一位经验丰富、富有激情的教育工作者，对操作系统领域有着深厚的知识和独到的见解。他在整个编写过程中提供了宝贵的指导和建议。同时，我们要感谢中国科学院大学《操作系统》课程实验课指导书的编写团队。他们经过精心的策划和努力，为学生们提供了一份详尽而系统的指导材料。这个团队的辛勤工作和专业知识为本手册的编写提供了坚实的基础。

我们希望通过本手册，能够帮助您更好地理解和应用操作系统内核开发的关键概念和技术。我们相信，通过实验的实践和理论的结合，您将能够获得宝贵的经验和深入的学习成果。

最后，我们衷心希望本手册能够对您的学习和实践产生积极的影响。如果您在使用过程中遇到任何问题或困惑，我们鼓励您及时寻求指导和支持。

再次感谢董恺老师和中国科学院大学《操作系统》课程实验课指导书的编写团队的辛勤付出!

祝您在《openEuler 内核开发实践》课程实践中取得优异的成绩!

敬祝学习愉快!

目录

1 *openEuler* 操作系统安装与内核编译

1.1 任务 1：openEuler 操作系统安装（30min）

1.1.1 任务描述

1. 下载最新版本的 openEuler 操作系统；将其安装至树莓派 4B 上。
2. 获取树莓派中 openEuler 系统的 IP 地址，使用 ssh 远程登录。

1.1.2 审核要求

1. 在树莓派 4B 中成功安装 openEuler 操作系统，使用用户名/密码正常登陆。
2. 使用 ssh 远程登录成功。
3. 提交相关流程截图。

1.1.3 操作指南

1. 树莓派刷机
   (a) 实验环境
   - windows10
   - 树莓派 4B/4G RAM
   - 16G 以上的 micro SD 卡

   (b) 获取树莓派 img 镜像
   - `https://repo.openeuler.org/openEuler-20.03-LTS-SP4/raspi_img`
   - 下载 img.xz 文件
   - 可通过 sha256sum 文件校验镜像完整性
   - 若下载的镜像扩展名是 iso，可直接将下载的镜像的扩展名重命名修改为 img。

   (c) 格式化 SD 卡
   - 使用 SDFormatter
   - 若 SD 卡之前未安装过镜像，盘符正常只有一个，直接格式化即可；
   - 如果若 SD 卡之前安装过镜像，则会出现多个盘符，格式化带有容量标记的即可。
   - 格式化结果为一个盘符，存储占用为空

   (d) 写入 SD 卡
   - 使用管理员身份运行 Win32 Disk Imager
   - 将镜像写入格式化后的磁盘
   - 结果会产生多个盘符

   (e) 分区扩容
   - 使用时的系统大小为 boot 盘的大小，可调整分区为 boot 盘扩容。

- 参考https://gitee.com/openeuler/raspberrypi/blob/master/documents/%E6%A0%91%E8%8E%93%E6%B4%BE%E4%BD%BF%E7%94%A8.md

2. 启用树莓派

(a) 启动树莓派

(b) PC 下载 SSH 链接工具

(c) 链接显示器获取树莓派 IP

- 使用 HDMI 视频输出树莓派界面
- 使用`uname -a`命令查看 IP 信息

(d) 同一局域网下获取树莓派 IP

- 树莓派与 windows10 的 PC 有线接入同一个路由器
- PC 端登入路由，查看 Pi4B IP

(e) 有线链接获取树莓派 IP 方法：

- 通过网线将树莓派与 PC 链接
- PC 端通过命令行使用`arp -a`命令查看局域网所有 IP 信息
- 在 PC 端网络连接设置界面将网络与树莓派连接共享
- 再次使用`arp -a`命令，对比 IP 变化
- 新增的内容包含树莓派 IP 地址

(f) ssh 连接命令`ssh root@[Pi4B IP]`

(g) openEuler 的用户名/密码一般是：root/openeuler

1.2 任务 2：openEuler 内核编译与安装（30min）

1.2.1 任务描述

1. 下载 openEuler-20.03-LTS-SP3 版本镜像对应的内核源码，编译内核源码。
2. 编译完成后安装/更新内核。
3. 由于内核编译过程耗时较长，可在编译的同时进行 [任务 3] 的实验内容。

1.2.2 审核要求

1. 正确编译内核源码，并完成安装。
2. 提交新旧内核版本的截图。

1.2.3 操作指南

1. 查看原始系统信息 (截图留存)

```
# uname -a  # 查看内核版本信息
```

2. 安装文件传输工具

```
# dnf install lrzsz
```

- 该工具可以较为方便的在 PC 与树莓派之间传输文件，有助于调试与后续实验。
- 若cz和rz命令无法使用，尝试更换支持此功能的 SSH 链接工具，如 XShell。
- 该工具不可传输大文件

3. 系统备份

```
# cd ~
# dnf install tar
# tar czvf boot_origin.tgz /boot/
# sz boot_origin.tgz  #将备份文件发送到本地(可选)
```

4. 注册账号、密钥

(a) 注册 gitee 账号

- https://gitee.com/

(b) 生成并部署 SSH key

- http://git.mydoc.io/?t=154712

(c) 测试 ssh 是否配置成功

```
# ssh -T git@gitee.com
```

- 如果未部署 SSH key 可以选择本地传输方式：

  (a) 登录后将内核源码压缩包下载至 PC。

(b) `https://gitee.com/openeuler/raspberrypi-kernel/tree/openEuler-20.03-LTS/`

(c) 树莓派使用`rz`命令将 PC 上压缩包传输至树莓派。

5. 内核源码下载

```
# dnf install wget
# wget https://gitee.com/openeuler/raspberrypi-kernel/repository/archive/openEuler-20.03-LTS.zip
# unzip openEuler-20.03-LTS.zip
# cd raspberrypi-kernel
```

- `wget`是一个在网络上进行下载的软件
- 网站需要在 SSH 配置成功前提下才可下载
- 网站在 PC 端需要登录后才可打开
- **需要使用 raspberrypi-kernel 目录下的树莓派内核**，不要使用 kernel 仓库中的内核，否则本实验不会报错但是后续实验会无法进行。

6. 编译内核

在根目录下，输入：

```
# dnf install gcc bison flex openssl-devel bc
# make openeuler-raspi_defconfig
# make -j4 Image modules dtbs #耗时长
# make modules_install
```

7. 安装、升级内核

```
# cp arch/arm64/boot/Image /boot/kernel8.img
# cp arch/arm64/boot/dts/broadcom/*.dtb /boot/
# cp arch/arm64/boot/dts/overlays/*.dtb* /boot/overlays/
# cp arch/arm64/boot/dts/overlays/README /boot/overlays/
```

8. 重启系统

```
# reboot
# uname -a  # 查看内核版本信息
Linux openEuler 4.19.90 #1 SMP PREEMPT Mon May 31 19:27:46
CST 2021 aarch64 aarch64 aarch64 GNU/Linux
```

9. 问题解决

- 如果启动失败，请把备份好的 boot_origin.tgz 解压覆盖到/boot 中并重启
- 如果编译操作系统过程中提示变量重命名，可尝试对 gcc 进行版本降级，或者修改内核源码，将二次定义变量行删除

1.3 任务 3：内核模块编程（30min）

1.3.1 任务描述

1. 编写内核模块，功能是打印“hello,world!”字符串。
2. 编写对应 Makefile 文件，并使用 make 编译上述内核模块。
3. 手动加载内核模块，查看加载内容。
4. 手动卸载上述内核模块。

1.3.2 审核要求

1. 正确编写满足功能的源文件，正确编译。
2. 正常加载、卸载内核模块；且内核模块功能满足任务所述。
3. 提交相关源码与运行截图。

1.3.3 操作指南

1. 内核预编译

```
# cd ~\raspberrypi-kernel
# make openeuler-raspi_defconfig # 加载内核配置
# make prepare && make scripts # 内核预编译（耗时短）
```

2. 创建路径、.c 文件、Makefile 文件

```
# cd ~\raspberrypi-kernel
# mkdir labwork_1.3
# cd labwork_1.3
# vim helloworld.c
# vim Makefile
```

3. 使用 vim
   - 进入 vim 后，默认进入命令模式
   - 命令模式依次键入 [:][q][Enter] 退出 vim
   - 命令模式按 [i] 进入插入模式，随光标键入内容
   - 插入模式按 [esc] 进入命令模式
   - 命令模式依次键入 [:][w][Enter] 保存，或 [:wq] 保存退出
   - 其余模式及操作请自行查阅资料
4. 编辑 helloworld.c

```
1 #include<linux/module.h>
2 MODULE_LICENSE("GPL");
3
4 int __init hello_init(void) {
5     printk("hello␣init\n");
```

```
    printk("hello,world!\n");
    return 0;
}
void __exit hello_exit(void) {
    printk("hello␣exit\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

5. 编辑 Makefile

```
ifneq ($(KERNELRELEASE),)
    obj-m := helloworld.o
else
    KERNELDIR ?=/root/raspberrypi-kernel
    PWD := $(shell pwd)
default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
.PHONY:clean
clean:
    -rm *.mod.c *.o *.order *.symvers *.ko
```

6. 执行 make 编译源码

```
[root@openEuler labwork_1.3]# make
make -C /root/raspberrypi-kernel M=/root/raspberrypi-
kernel/labwork_1.3 modules
make[1]: Entering directory '/root/raspberrypi-kernel'
  Building modules, stage 2.
  MODPOST 1 modules
make[1]: Leaving directory '/root/raspberrypi-kernel'
[root@openEuler labwork_1.3]#
```

7. 进行模块加载、查看、卸载

```
# dmesg -c  # 查看并清空消息
# dmesg  # 消息已清空
# insmod helloworld.ko # 加载
# lsmod | grep hello # 查看模块，结果中查找hello字串
# dmesg -c
# rmmod helloworld.ko # 卸载
# lsmod | grep hello
# dmesg
```

2 内存管理

2.1 任务 1：使用 kmalloc 分配 1KB，8KB 的内存，并打印指针地址（20min）

2.1.1 任务描述

1. 使用 kmalloc 分配 1KB，8KB 的内存，打印指针地址；
2. 查看已分配的内存，根据机器是 32 位或 64 位的情况，分析地址落在的区域。

2.1.2 审核要求

1. 正确编写满足功能的源文件，正确编译。
2. 正常加载、卸载内核模块；且内核模块功能满足任务所述。
3. 提交相关源码、运行截图以及内存分配情况的解释。

2.1.3 操作指南

1. 编辑 kmalloc.c

```c
#include <linux/module.h>
#include <linux/slab.h>

MODULE_LICENSE("GPL");

unsigned char *kmallocmem1;
unsigned char *kmallocmem2;

static int __init mem_module_init(void){
    printk("Start␣kmalloc!\n");
    kmallocmem1 = (unsigned char *)kmalloc(1024, GFP_KERNEL);
    if (kmallocmem1 != NULL){
        printk(KERN_ALERT "kmallocmem1␣addr␣=␣%lx\n", (unsigned long)kmallocmem1);
    }
    else{
        printk("Failed␣to␣allocate␣kmallocmem1!\n");
    }
    kmallocmem2 = (unsigned char *)kmalloc(8192, GFP_KERNEL);
    if (kmallocmem2 != NULL){
        printk(KERN_ALERT "kmallocmem2␣addr␣=␣%lx\n", (unsigned long)kmallocmem2);
    }
    else{
        printk("Failed␣to␣allocate␣kmallocmem2!\n");
    }
    return 0;
}

static void __exit mem_module_exit(void){
    kfree(kmallocmem1);
    kfree(kmallocmem2);
    printk("Exit␣kmalloc!\n");
}

module_init(mem_module_init);
```

```
35 module_exit(mem_module_exit);
```

2. 编辑 Makefile

```
1  ifneq ($(KERNELRELEASE),)
2      obj-m := kmalloc.o
3  else
4      KERNELDIR ?=/root/raspberrypi-kernel
5      PWD := $(shell pwd)
6  default:
7      $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
8  endif
9  .PHONY:clean
10 clean:
11     -rm *.mod.c *.o *.order *.symvers *.ko
```

3. 执行 make 编译源码

4. 进行模块加载、查看、卸载

5. 判断地址位于内核空间还是用户空间

   - 使用arch命令查询 CPU 位数
   - 通过getconf PAGE_SIZE命令获得页大小
   - 打开arch/arm64/configs/openeuler-raspi_defconfig文件，其中CONFIG_PGTABLE_LEVELS=4表明页表级数。
   - 通过获得的信息，在官方文件https://www.kernel.org/doc/Documentation/arm64/memory.txt 中检索得到用户内存与内核内存的范围。

2.2 任务 2：使用 vmalloc 分别分配 8KB、1MB、64MB 的内存，打印指针地址（20min）

2.2.1 任务描述

1. 使用 vmalloc 分配 8KB、1MB、64MB 的内存，打印指针地址；
2. 查看已分配的内存，根据机器是 32 位或 64 位的情况，分析地址落在的区域。

2.2.2 审核要求

1. 正确编写满足功能的源文件，正确编译。
2. 正常加载、卸载内核模块；且内核模块功能满足任务所述。
3. 提交相关源码、运行截图以及内存分配情况的解释。

2.2.3 操作指南

1. 编辑 vmalloc.c

```c
#include <linux/module.h>
#include <linux/vmalloc.h>

MODULE_LICENSE("GPL");

unsigned char *vmallocmem1;
unsigned char *vmallocmem2;
unsigned char *vmallocmem3;

static int __init mem_module_init(void)
{
    printk("Start␣vmalloc!\n");
    vmallocmem1 = (unsigned char *)vmalloc(8192);
    if (vmallocmem1 != NULL)
    {
        printk("vmallocmem1␣addr␣=␣%lx\n", (unsigned long)vmallocmem1);
    }
    else
    {
        printk("Failed␣to␣allocate␣vmallocmem1!\n");
    }
    vmallocmem2 = (unsigned char *)vmalloc(1048576);
    if (vmallocmem2 != NULL)
    {
        printk("vmallocmem2␣addr␣=␣%lx\n", (unsigned long)vmallocmem2);
    }
    else
    {
        printk("Failed␣to␣allocate␣vmallocmem2!\n");
    }
    vmallocmem3 = (unsigned char *)vmalloc(67108864);
    if (vmallocmem3 != NULL)
    {
        printk("vmallocmem3␣addr␣=␣%lx\n", (unsigned long)vmallocmem3);
    }
    else
```

```
37      {
38          printk("Failed␣to␣allocate␣vmallocmem3!\n");
39      }
40      return 0;
41  }
42
43  static void __exit mem_module_exit(void)
44  {
45      vfree(vmallocmem1);
46      vfree(vmallocmem2);
47      vfree(vmallocmem3);
48      printk("Exit␣vmalloc!\n");
49  }
50
51  module_init(mem_module_init);
52  module_exit(mem_module_exit);
```

2. 编辑 Makefile

```
1  ifneq ($(KERNELRELEASE),)
2      obj-m := vmalloc.o
3  else
4      KERNELDIR ?=/root/raspberrypi-kernel
5      PWD := $(shell pwd)
6  default:
7      $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
8  endif
9  .PHONY:clean
10 clean:
11     -rm *.mod.c *.o *.order *.symvers *.ko
```

3. 执行 make 编译源码

4. 进行模块加载、查看、卸载

5. 判断地址位于内核空间还是用户空间

3 进程管理

3.1 任务 1：创建并运行内核线程（15min）

3.1.1 任务描述

1. 编写内核模块，创建一个内核线程；并在模块退出时杀死该线程。
2. 加载、卸载模块并查看模块打印信息。

3.1.2 审核要求

1. 正确编写满足功能的源文件，正确编译。
2. 正常加载、卸载内核模块；且内核模块功能满足任务所述。
3. 提交相关源码与运行截图。

3.1.3 操作指南

1. 编辑 kthread.c

```c
#include <linux/kthread.h>
#include <linux/module.h>
#include <linux/delay.h>

MODULE_LICENSE("GPL");

#define BUF_SIZE 20

static struct task_struct *myThread = NULL;

static int print(void *data)
{
    while (!kthread_should_stop())
    {
        printk("New␣kthread␣is␣running.␣2sec␣per␣msg.\n");
        msleep(2000);
    }
    return 0;
}

static int __init kthread_init(void)
{
    printk("Create␣kernel␣thread!\n");
    myThread = kthread_run(print, NULL, "new_kthread");
    return 0;
}

static void __exit kthread_exit(void)
{
    printk("Kill␣new␣kthread.\n");
    if (myThread)
        kthread_stop(myThread);
}

```

```
35 module_init(kthread_init);
36 module_exit(kthread_exit);
```

2. 编辑 Makefile

```
1  ifneq ($(KERNELRELEASE),)
2      obj-m := kthread.o
3  else
4      KERNELDIR ?=/root/raspberrypi-kernel
5      PWD := $(shell pwd)
6  default:
7      $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
8  endif
9  .PHONY:clean
10 clean:
11     -rm *.mod.c *.o *.order *.symvers *.ko
```

3. 执行 make 编译源码

4. 进行模块加载、查看、卸载

3.2 任务 2：打印输出当前系统 CPU 负载情况（20min）

3.2.1 任务描述

1. 编写一个内核模块，实现读取系统一分钟内的 CPU 负载。
2. 加载、卸载模块并查看模块打印信息。

3.2.2 审核要求

1. 正确编写满足功能的源文件，正确编译。
2. 正常加载、卸载内核模块；且内核模块功能满足任务所述。
3. 提交相关源码与运行截图。

3.2.3 操作指南

1. 编辑cpu_loadavg.c

```c
#include <linux/module.h>
#include <linux/fs.h>

MODULE_LICENSE("GPL");

char tmp_cpu_load[5] = {'\0'};

static int get_loadavg(void)
{
    struct file *fp_cpu;
    loff_t pos = 0;
    char buf_cpu[10];
    fp_cpu = filp_open("/proc/loadavg", O_RDONLY, 0);
    if (IS_ERR(fp_cpu))
    {
        printk("Failed to open loadavg file!\n");
        return -1;
    }
    kernel_read(fp_cpu, buf_cpu, sizeof(buf_cpu), &pos);
    strncpy(tmp_cpu_load, buf_cpu, 4);
    filp_close(fp_cpu, NULL);
    return 0;
}

static int __init cpu_loadavg_init(void)
{
    printk("Start cpu_loadavg!\n");
    if (0 != get_loadavg())
    {
        printk("Failed to read loadarvg file!\n");
        return -1;
    }
    printk("The cpu loadavg in one minute is: %s\n", tmp_cpu_load);
    return 0;
}

```

```
37 static void __exit cpu_loadavg_exit(void)
38 {
39     printk("Exit␣cpu_loadavg!\n");
40 }
41
42 module_init(cpu_loadavg_init);
43 module_exit(cpu_loadavg_exit);
```

2. 编辑 Makefile

```
1  ifneq ($(KERNELRELEASE),)
2      obj-m := cpu_loadavg.o
3  else
4      KERNELDIR ?= /root/raspberrypi-kernel
5      PWD := $(shell pwd)
6  default:
7      $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
8  endif
9  .PHONY:clean
10 clean:
11     -rm *.mod.c *.o *.order *.symvers *.ko
```

3. 执行 make 编译源码

4. 进行模块加载、查看、卸载

3.3 任务 3：打印输出当前处于运行状态的进程的 PID 和名字（15min）

3.3.1 任务描述

1. 编写一个内核模块，打印当前系统处于运行状态的进程的 PID 和名字。
2. 加载、卸载模块并查看模块打印信息。

3.3.2 审核要求

1. 正确编写满足功能的源文件，正确编译。
2. 正常加载、卸载内核模块；且内核模块功能满足任务所述。
3. 提交相关源码与运行截图。

3.3.3 操作指南

1. 编辑process_info.c

```c
#include <linux/module.h>
#include <linux/sched/signal.h>
#include <linux/sched.h>

MODULE_LICENSE("GPL");

struct task_struct *p;

static int __init process_info_init(void)
{
    printk("Start process_info!\n");
    for_each_process(p)
    {
        if (p->state == 0)
            printk("1)name:%s\t2)pid:%d\t3)state:%ld\n", p->comm, p->pid, p->state);
    }
    return 0;
}

static void __exit process_info_exit(void)
{
    printk("Exit process_info!\n");
}

module_init(process_info_init);
module_exit(process_info_exit);
```

2. 编辑 Makefile

```
ifneq ($(KERNELRELEASE),)
    obj-m := process_info.o
else
    KERNELDIR ?= /root/raspberrypi-kernel
    PWD := $(shell pwd)
default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
```

```
8  endif
9  .PHONY:clean
10 clean:
11     -rm *.mod.c *.o *.order *.symvers *.ko
```

3. 执行 make 编译源码

4. 进行模块加载、查看、卸载

3.4 任务 4：使用 cgroup 实现限制 CPU 核数（20min）

3.4.1 任务描述

1. 使用 cgroup 实现限制 CPU 核数；
2. 编写一个简单的 c 源程序，实现无线循环/死循环，使其占用某一进程（默认情况下会使得 cpu 资源消耗在 100）；
3. 使用 cgexec 与 taskset 测试上述限制操作是否成功。

3.4.2 审核要求

1. 正确、成功地限制 CPU 核数。
2. 提交每一步操作以及结果显示的截图。

3.4.3 操作指南

1. 挂载 tmpfs 格式的 cgroup 文件夹

```
# mkdir /cgroup
# mount -t tmpfs tmpfs /cgroup
# cd /cgroup
```

2. 挂载 cpuset 管理子系统

```
# mkdir cpuset
# mount -t cgroup -o cpuset cpuset /cgroup/cpuset #挂载cpuset子系统
# cd cpuset
# mkdir mycpuset  #创建一个控制组，删除用 rmdir 命令
# cd mycpuset
```

3. 设置 cpu 核数

```
# cat cpuset.mems
# echo 0 > cpuset.mems
# cat cpuset.cpus
# echo 0-2 > cpuset.cpus
# cat cpuset.mems
# cat cpuset.cpus
# cd mycpuset
```

4. 使用死循环 C 源文件`while_long.c` 测试验证

3.5 任务 5：使用 cgroup 实现不允许访问 U 盘（15min）

3.5.1 任务描述

1. 使用 cgroup 实现不允许访问 U 盘。
2. 使用 cgexec 与 dd 命令验证上述限制操作是否成功。

3.5.2 审核要求

1. 正确、成功地限制 U 盘访问。
2. 提交每一步操作以及结果显示的截图。

3.5.3 操作指南

1. 将 U 盘插入树莓派，使用 fdisk -l 获取该 U 盘的盘符
2. 调用 shell 命令“ls -l”获取设备号
3. 将 U 盘挂载到当前系统中

```
# mkdir /usb
# mount /dev/sda4 /usb
```

4. 挂载设备管理 devices 子系统

```
# cd /cgroup/
# mkdir devices
# mount -t cgroup -o devices devices /cgroup/devices #挂载devices子系统
# cd /cgroup/devices
# mkdir mydevices  # 创建mydevices控制组
# cd mydevices
```

5. 设置拒绝 U 盘访问

```
# echo 'a␣8:4␣rwm' > /cgroup/devices/mydevices/devices.deny
```

6. 测试验证

3.6 相关问题解决

1. 平均负载是指什么？
   系统平均负载被定义为在特定时间间隔内运行队列中的平均进程数。

2. 通过 screen 组件创建新终端

   - 创建一个名字为 XX 的终端:`screen -S XX`
   - 退出: 按 Ctrl+a，然后再按 d
   - 查看所有的终端：`screen -ls`
   - 进入终端界面:`screen -r name-or-id`
   - 删除:`screen -S name-or-id -X quit`

3. 如果插入 U 盘后`fdisk -l`命令找不到设备

   - 检查 U 盘是否出现问题
   - 检查树莓派是否出现问题
   - 检查操作系统是否出现问题
     操作系统问题可参考 Issue: `https://gitee.com/openeuler/raspberrypi/issues/I45FML`
     可能是内核版本问题造成的，尝试使用新内核

4 中断和异常管理

4.1 任务 1：使用 tasklet 实现打印 helloworld（20min）

4.1.1 任务描述

1. 编写内核模块，使用 tasklet 实现打印 helloworld。
2. 加载、卸载模块并查看模块打印信息。

4.1.2 审核要求

1. 正确编写满足功能的源文件，正确编译。
2. 正常加载、卸载内核模块；且内核模块功能满足任务所述。
3. 提交相关源码与运行截图。

4.1.3 操作指南

1. 编辑tasklet_interrupt.c

```c
#include <linux/module.h>
#include <linux/interrupt.h>

MODULE_LICENSE("GPL");

static struct tasklet_struct my_tasklet;

static void tasklet_handler(unsigned long data)//处理函数
{
    printk("Hello␣World!␣tasklet␣is␣working...\n");
}

static int __init mytasklet_init(void)
{
    printk("Start␣tasklet␣module...\n");
    tasklet_init(&my_tasklet, tasklet_handler, 0);//创建tasklet
    tasklet_schedule(&my_tasklet);//执行中断
    return 0;
}

static void __exit mytasklet_exit(void)
{
    tasklet_kill(&my_tasklet);//移除tasklet
    printk("Exit␣tasklet␣module...\n");
}

module_init(mytasklet_init);
module_exit(mytasklet_exit);
```

2. 编辑 Makefile

```
ifneq ($(KERNELRELEASE),)
    obj-m := tasklet_interrupt.o
```

```
3  else
4      KERNELDIR ?= /root/raspberrypi-kernel
5      PWD := $(shell pwd)
6  default:
7      $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
8  endif
9  .PHONY:clean
10 clean:
11     -rm *.mod.c *.o *.order *.symvers *.ko
```

3. 执行 make 编译源码

4. 进行模块加载、查看、卸载

4.2 任务 2：用工作队列实现周期打印 helloworld（25min）

4.2.1 任务描述

1. 编写一个内核模块程序，用工作队列实现周期打印 helloworld。
2. 加载、卸载模块并查看模块打印信息。

4.2.2 审核要求

1. 正确编写满足功能的源文件，正确编译。
2. 正常加载、卸载内核模块；且内核模块功能满足任务所述。
3. 提交相关源码与运行截图。

4.2.3 操作指南

1. 编辑workqueue_test.c

```c
#include <linux/module.h>
#include <linux/workqueue.h>
#include <linux/delay.h>

MODULE_LICENSE("GPL");
static struct workqueue_struct *queue = NULL;
static struct delayed_work mywork;
static int i = 0;

// work handle
void work_handle(struct work_struct *work)
{
    printk(KERN_ALERT "Hello World! From 09021230 Yablin SUN\n");
}

static int __init timewq_init(void)
{
    printk(KERN_ALERT "Start workqueue_test module.");
    queue = create_singlethread_workqueue("workqueue_test");
    if (queue == NULL)
    {
        printk(KERN_ALERT "Failed to create workqueue_test!\n");
        return -1;
    }
    INIT_DELAYED_WORK(&mywork, work_handle);
    for (; i <= 3; i++)
    {
        queue_delayed_work(queue, &mywork, 1 * HZ);
        ssleep(2);
    }
    return 0;
}

static void __exit timewq_exit(void)
{
    flush_workqueue(queue);
```

```
37      destroy_workqueue(queue);
38      printk(KERN_ALERT "Exit␣workqueue_test␣module.");
39  }
40
41  module_init(timewq_init);
42  module_exit(timewq_exit);
```

2. 编辑 Makefile

```
1   ifneq ($(KERNELRELEASE),)
2       obj-m := workqueue_test.o
3   else
4       KERNELDIR ?= /root/raspberrypi-kernel
5       PWD := $(shell pwd)
6   default:
7       $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
8   endif
9   .PHONY:clean
10  clean:
11      -rm *.mod.c *.o *.order *.symvers *.ko
```

3. 执行 make 编译源码

4. 进行模块加载、查看、卸载

4.3 任务 3：编写一个信号捕获程序，捕获终端按键信号（25min）

4.3.1 任务描述

1. 在用户态编写一个信号捕获程序，捕获终端按键信号。
2. 编译上述程序后运行，在终端输入按键信号，查看输出信息。

4.3.2 审核要求

1. 正确编写满足功能的源文件，正确编译。
2. 提交相关源码与运行截图。

4.3.3 操作指南

1. 编辑catch_signal.c

```c
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

void signal_handler(int sig)
{
    switch(sig){
        case SIGINT:
            printf("\nGet␣a␣signal:SIGINT.␣You␣pressed␣ctrl+c.\n");
            break;
        case SIGQUIT:
            printf("\nGet␣a␣signal:SIGQUIT.␣You␣pressed␣ctrl+\\.\n");
            break;
        case SIGTSTP:
            printf("\nGet␣a␣signal:SIGHUP.␣You␣pressed␣ctrl+z.\n");
            break;
    }
    exit(0);
}

int main()
{
    printf("Current␣process␣ID␣is␣%d\n", getpid());
    signal(SIGINT, signal_handler);
    signal(SIGQUIT, signal_handler);
    signal(SIGTSTP, signal_handler);
    for(;;);
}
```

2. gcc 编译源码

3. 捕获信号

5 内核时间管理

5.1 任务 1：调用内核时钟接口打印当前时间（20min）

5.1.1 任务描述

1. 编写内核模块，调用内核时钟接口，打印出系统当前时间。格式示例：2020-03-09 11:54:31；
2. 加载、卸载模块并查看模块打印信息。

5.1.2 审核要求

1. 正确编写满足功能的源文件，正确编译。
2. 正常加载、卸载内核模块；且内核模块功能满足任务所述。
3. 提交相关源码与运行截图。

5.1.3 操作指南

1. 编辑current_time.c

```c
#include <linux/module.h>
#include <linux/time.h>
#include <linux/rtc.h>

MODULE_LICENSE("GPL");

ktime_t k_time;
struct rtc_time tm;

static int __init currenttime_init(void)
{
    int year, mon, day, hour, min, sec;
    printk("Start current_time module...\n");
    k_time = ktime_get_real();
    tm = rtc_ktime_to_tm(k_time);
    year = tm.tm_year + 1900;
    mon = tm.tm_mon + 1;
    day = tm.tm_mday;
    hour = tm.tm_hour + 8;
    min = tm.tm_min;
    sec = tm.tm_sec;
    printk("Current time is: %d-%02d-%02d %02d:%02d:%02d \t printed from SYL\n", year, mon, day, hour, min, sec);
    return 0;
}

static void __exit currenttime_exit(void)
{
    printk("Exit current_time module...\n");
}

module_init(currenttime_init);
module_exit(currenttime_exit);
```

2. 编辑 Makefile

```
ifneq ($(KERNELRELEASE),)
    obj-m := current_time.o
else
    KERNELDIR ?= /root/raspberrypi-kernel
    PWD := $(shell pwd)
default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
.PHONY:clean
clean:
    -rm *.mod.c *.o *.order *.symvers *.ko
```

3. 执行 make 编译源码

4. 进行模块加载、查看、卸载

5.2 任务 2：编写 timer，在特定时刻打印 hello,world（25min）

5.2.1 任务描述

1. 编写内核模块程序，实现一个 timer，该定时器延时 10 秒后打印“hello,world”。
2. 加载、卸载模块并查看模块打印信息。验证超时时间并截图。

5.2.2 审核要求

1. 正确编写满足功能的源文件，正确编译。
2. 正常加载、卸载内核模块；且内核模块功能满足任务所述。
3. 提交相关源码与运行截图。

5.2.3 操作指南

1. 编辑timer_example.c

```c
#include <linux/module.h>
#include <linux/timer.h>

MODULE_LICENSE("GPL");

struct timer_list timer;

void print(struct timer_list *timer)
{
    printk("hello,world!\t␣printed␣from␣SYL\n");
}

static int __init timer_init(void)
{
    printk("Start␣timer_example␣module...\n");
    timer.expires = jiffies + 10 * HZ;
    timer.function = print;
    add_timer(&timer);
    return 0;
}

static void __exit timer_exit(void)
{
    printk("Exit␣timer_example␣module...\n");
}

module_init(timer_init);
module_exit(timer_exit);
```

2. 编辑 Makefile

```
ifneq ($(KERNELRELEASE),)
    obj-m := timer_example.o
else
    KERNELDIR ?= /root/raspberrypi-kernel
    PWD := $(shell pwd)
```

```
6  default:
7      $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
8  endif
9  .PHONY:clean
10 clean:
11     -rm *.mod.c *.o *.order *.symvers *.ko
```

3. 执行 make 编译源码

4. 进行模块加载、查看、卸载

5.3 任务 3：调用内核时钟接口，监控累加计算代码的运行时间（40min）

5.3.1 任务描述

1. 调用内核时钟接口，编写内核模块，监控实现累加计算 sum=1+2+3+...+100000 所花时间。
2. 加载、卸载模块并查看模块打印信息。

5.3.2 审核要求

1. 正确编写满足功能的源文件，正确编译。
2. 正常加载、卸载内核模块；且内核模块功能满足任务所述。
3. 提交相关源码与运行截图。

5.3.3 操作指南

1. 编辑sum_time.c

```c
#include <linux/module.h>
#include <linux/time.h>

MODULE_LICENSE("GPL");

#define NUM 100000

static long sum(int num)
{
    int i;
    long total = 0;
    for (i = 1; i <= NUM; i++)
        total = total + i;
    return total;
}

static int __init sum_init(void)
{
    ktime_t startTime = 0;
    ktime_t endTime = 0;

    long time_cost;
    long s;

    printk("Start␣sum_time␣module...\n");

    startTime = ktime_get_real();
    s = sum(NUM);
    endTime = ktime_get_real();

    printk("The␣start␣time␣is:␣%lld␣ns␣\n", startTime);
    printk("The␣sum␣of␣1␣to␣%d␣is:␣%ld\n", NUM, s);
    printk("The␣end␣time␣is:␣%lld␣ns␣\n", endTime);

    printk("The␣cost␣time␣of␣sum␣from␣1␣to␣%d␣is:␣%lld␣ns␣\n", NUM, endTime - startTime);
    return 0;
```

```
}

static void __exit sum_exit(void)
{
    printk("Exit␣sum_time␣module...\n");
}

module_init(sum_init);
module_exit(sum_exit);
```

2. 编辑 Makefile

```
ifneq ($(KERNELRELEASE),)
    obj-m := sum_time.o
else
    KERNELDIR ?= /root/raspberrypi-kernel
    PWD := $(shell pwd)
default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
.PHONY:clean
clean:
    -rm *.mod.c *.o *.order *.symvers *.ko
```

3. 执行 make 编译源码

4. 进行模块加载、查看、卸载

5.4 相关问题解决

1. 编译报错`error: implicit declaration of function 'do_gettimeofday'`

   - `do_gettimeofday()`函数在 linux kernel 5.x 之后淘汰
   - 修改以下注释部分代码

```
//struct timeval tv;
//struct rtc_time tm;
ktime_t k_time;
struct rtc_time tm;

//do_gettimeofday(&tv);
//rtc_time_to_tm(tv.tv_sec, &tm);
k_time = ktime_get_real();
tm = rtc_ktime_to_tm(k_time);
```

   - 操作系统的时间可能因为没有校准而显示与当前时间不同。
     可以通过 ‘date‘命令查看 Linux 系统时间，并对比是否一致。

6 设备管理

6.1 任务 1：编写 USB 设备驱动程序（40min）

6.1.1 任务描述

1. 参考内核源码中的 drivers/usb/usb-skeleton.c 文件，编写一个 USB 探测驱动程序
2. 加载、卸载模块并查看模块打印信息。

6.1.2 审核要求

1. 正确编写满足功能的源文件，正确编译。
2. 正常加载、卸载内核模块；且内核模块功能满足任务所述。
3. 提交相关源码与运行截图。

6.1.3 操作指南

1. 编辑usb_detect.c注意修改设备号

```c
/*
 * USB Detect driver
 * Written by 09021230 SYL
 * This driver is based on the 2.6.3 version of drivers/usb/usb-skeleton.c
 */

#include <linux/kernel.h>
#include <linux/errno.h>
#include <linux/slab.h>
#include <linux/module.h>
#include <linux/kref.h>
#include <linux/uaccess.h>
#include <linux/usb.h>
#include <linux/mutex.h>


/* Define these values to match your devices */
#define USB_DETECT_VENDOR_ID 0x0781
#define USB_DETECT_PRODUCT_ID 0x5591

/* table of devices that work with this driver */
static const struct usb_device_id usbdetect_table[] = {
    { USB_DEVICE(USB_DETECT_VENDOR_ID, USB_DETECT_PRODUCT_ID) },
    { }                 /* Terminating entry */
};
MODULE_DEVICE_TABLE(usb, usbdetect_table);


/* Get a minor range for your devices from the usb maintainer */
#define USB_DETECT_MINOR_BASE 192

#define WRITES_IN_FLIGHT 8
/* arbitrarily chosen */

```

```
35 /* Structure to hold all of our device specific stuff */
36 struct usb_detect {
37     struct usb_device *udev;       /* the usb device for this device */
38     struct usb_interface *interface;  /* the interface for this device */
39     struct semaphore limit_sem;    /* limiting the number of writes in progress */
40     struct usb_anchor submitted;   /* in case we need to retract our submissions */
41     struct urb    *bulk_in_urb;    /* the urb to read data with */
42     unsigned char       *bulk_in_buffer; /* the buffer to receive data */
43     size_t        bulk_in_size;    /* the size of the receive buffer */
44     size_t        bulk_in_filled;  /* number of bytes in the buffer */
45     size_t        bulk_in_copied;  /* already copied to user space */
46     __u8          bulk_in_endpointAddr; /* the address of the bulk in endpoint */
47     __u8          bulk_out_endpointAddr; /* the address of the bulk out endpoint */
48     int       errors;       /* the last request tanked */
49     bool          ongoing_read;    /* a read is going on */
50     spinlock_t    err_lock;     /* lock for errors */
51     struct kref   kref;
52     struct mutex     io_mutex;     /* synchronize I/O with disconnect */
53     unsigned long    disconnected:1;
54     wait_queue_head_t bulk_in_wait;   /* to wait for an ongoing read */
55 };
56 #define to_detect_dev(d) container_of(d, struct usb_detect, kref)
57
58 static struct usb_driver usbdetect_driver;
59
60 //USB 拔出中断
61 static void usbdetect_delete(struct kref *kref){
62     struct usb_detect *dev = to_detect_dev(kref);
63
64     usb_free_urb(dev->bulk_in_urb);
65     usb_put_intf(dev->interface);
66     usb_put_dev(dev->udev);
67     kfree(dev->bulk_in_buffer);
68     kfree(dev);
69 }
70
71 static const struct file_operations usbdetect_fops = {};
72
73 /*
74  * usb class driver info in order to get a minor number from the usb core,
75  * and to have the device registered with the driver core
76  */
77 static struct usb_class_driver usbdetect_class = {
78     .name =    ">>>␣TEST␣<<<%d",
79     .fops =    &usbdetect_fops,
80     .minor_base = USB_DETECT_MINOR_BASE,
81 };
82
83 //探测函数
84 static int usbdetect_probe(struct usb_interface *interface, const struct usb_device_id *id){
85     struct usb_detect *dev;
86     struct usb_endpoint_descriptor *bulk_in, *bulk_out;
87     int retval;
88
89     /* allocate memory for our device state and initialize it */
90     dev = kzalloc(sizeof(*dev), GFP_KERNEL);
91     if (!dev)
```

```
92          return -ENOMEM;
93
94      kref_init(&dev->kref);
95      sema_init(&dev->limit_sem, WRITES_IN_FLIGHT);
96      mutex_init(&dev->io_mutex);
97      spin_lock_init(&dev->err_lock);
98      init_usb_anchor(&dev->submitted);
99      init_waitqueue_head(&dev->bulk_in_wait);
100
101     dev->udev = usb_get_dev(interface_to_usbdev(interface));
102     dev->interface = usb_get_intf(interface);
103
104     /* set up the endpoint information */
105     /* use only the first bulk-in and bulk-out endpoints */
106     retval = usb_find_common_endpoints(interface->cur_altsetting,
107             &bulk_in, &bulk_out, NULL, NULL);
108     if (retval) {
109         dev_err(&interface->dev,
110             "Could not find both bulk-in and bulk-out endpoints\n");
111         goto error;
112     }
113
114     dev->bulk_in_size = usb_endpoint_maxp(bulk_in);
115     dev->bulk_in_endpointAddr = bulk_in->bEndpointAddress;
116     dev->bulk_in_buffer = kmalloc(dev->bulk_in_size, GFP_KERNEL);
117     if (!dev->bulk_in_buffer) {
118         retval = -ENOMEM;
119         goto error;
120     }
121     dev->bulk_in_urb = usb_alloc_urb(0, GFP_KERNEL);
122     if (!dev->bulk_in_urb) {
123         retval = -ENOMEM;
124         goto error;
125     }
126
127     dev->bulk_out_endpointAddr = bulk_out->bEndpointAddress;
128
129     /* save our data pointer in this interface device */
130     usb_set_intfdata(interface, dev);
131
132     /* we can register the device now, as it is ready */
133     retval = usb_register_dev(interface, &usbdetect_class);
134     if (retval) {
135         /* something prevented us from registering this driver */
136         dev_err(&interface->dev,
137             "Not able to get a minor for this device.\n");
138         usb_set_intfdata(interface, NULL);
139         goto error;
140     }
141
142     /* let the user know what node this device is now attached to */
143     dev_info(&interface->dev,
144         "USB detect device now attached to USBdetect-%d",
145         interface->minor);
146     return 0;
147
148 error:
```

```
149     /* this frees allocated memory */
150     kref_put(&dev->kref, usbdetect_delete);
151 
152     return retval;
153 }
154 
155 //拔出函数
156 static void usbdetect_disconnect(struct usb_interface *interface){
157     struct usb_detect *dev;
158     int minor = interface->minor;
159 
160     dev = usb_get_intfdata(interface);
161     usb_set_intfdata(interface, NULL);
162 
163     /* give back our minor */
164     usb_deregister_dev(interface, &usbdetect_class);
165 
166     /* prevent more I/O from starting */
167     mutex_lock(&dev->io_mutex);
168     dev->disconnected = 1;
169     mutex_unlock(&dev->io_mutex);
170 
171     usb_kill_anchored_urbs(&dev->submitted);
172 
173     /* decrement our usage count */
174     kref_put(&dev->kref, usbdetect_delete);
175 
176     dev_info(&interface->dev, "USB␣detect␣#%d␣now␣disconnected", minor);
177 }
178 
179 static struct usb_driver usbdetect_driver = {
180     .name =    ">>>␣TEST␣<<<",
181     .probe =  usbdetect_probe,
182     .disconnect = usbdetect_disconnect,
183     .id_table = usbdetect_table,
184     .supports_autosuspend = 1,
185 };
186 
187 //模块信息
188 
189 MODULE_LICENSE("GPL␣v2");
190 
191 static int __init usb_detect_init(void){
192     int result;
193     printk("Start␣usb_detect␣module...");
194     /* register this driver with the USB subsystem */
195     result = usb_register(&usbdetect_driver);
196     if (result < 0) {
197         printk("usb_register␣failed.""Error␣number␣%d", result);
198         return -1;
199     }
200     return 0;
201 }
202 
203 static void __exit usb_detect_exit(void)
204 {
205     printk("Exit␣usb_detect␣module...");
```

```
206     /* deregister this driver with the USB subsystem */
207     usb_deregister(&usbdetect_driver);
208 }
209 
210 module_init(usb_detect_init);
211 module_exit(usb_detect_exit);
```

2. 编辑 Makefile

```
1  ifneq ($(KERNELRELEASE),)
2      obj-m := usb_detect.o
3  else
4      KERNELDIR ?= /root/raspberrypi-kernel
5      PWD := $(shell pwd)
6  default:
7      $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
8  endif
9  .PHONY:clean
10 clean:
11     -rm *.mod.c *.o *.order *.symvers *.ko
```

3. 执行 make 编译源码

4. 进行模块加载、查看、卸载

6.2 任务 2:编写内核模块测试硬盘的读写速率,并与 iozone 工具的测试结果比较(45min)

6.2.1 任务描述

1. 编写内核模块测试硬盘的读写速率，加载、卸载模块并查看模块打印信息。
2. 使用用户态下 iozone 工具测试硬盘的读写速率，注意：测试范围需包含与内核模块读写相同的文件大小和块大小。
3. 对比用户态和内核态下测试的读写速率，并作分析。

6.2.2 审核要求

1. 正确编写满足功能的源文件，正确编译。
2. 正常加载、卸载内核模块；且内核模块功能满足任务所述。
3. 正确安装使用 iozone 工具测试硬盘读写速率。（可能会比较耗时）
4. 提交相关源码与运行截图。

6.2.3 操作指南

1. 编辑`write_to_disk.c`

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/rtc.h>

#define buf_size 1024
#define write_times 524288

MODULE_LICENSE("GPL");

ktime_t startTime = 0;
ktime_t endTime = 0;

static int __init write_disk_init(void)
{
    struct file *fp_write;
    char buf[buf_size];
    int i;
    int write_time;
    loff_t pos;
    printk("Start␣write_to_disk␣module...\n");
    for(i = 0; i < buf_size; i++)
    {
        buf[i] = i + '0';
    }
    fp_write = filp_open("/home/tmp_file", O_RDWR | O_CREAT,0644);
    if (IS_ERR(fp_write)) {
        printk("Failed␣to␣open␣file...\n");
        return -1;
    }
    pos = 0;
```

```
    startTime = ktime_get_real();

    for(i = 0; i < write_times; i++) {
        kernel_write(fp_write, buf, buf_size, &pos);
    }
    endTime = ktime_get_real();
    filp_close(fp_write, NULL);
    write_time = (endTime - startTime) / 1000;
    printk(KERN_ALERT "Writing␣to␣file␣costs␣%d␣us\n", write_time);
    printk("Writing␣speed␣is␣%d␣M/s\n", buf_size * write_times / write_time);
    return 0;
}

static void __exit write_disk_exit(void)
{
    printk("Exit␣write_to_disk␣module...\n");
}

module_init(write_disk_init);
module_exit(write_disk_exit);
```

2. 编辑 Makefile

```
ifneq ($(KERNELRELEASE),)
    obj-m := write_to_disk.o
else
    KERNELDIR ?= /root/raspberrypi-kernel
    PWD := $(shell pwd)
default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
.PHONY:clean
clean:
    -rm *.mod.c *.o *.order *.symvers *.ko
```

3. 执行 make 编译源码

4. 进行模块加载、查看、卸载

5. 编辑`write_to_disk.c`

```
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/rtc.h>

#define buf_size 1024
#define read_times 524288

MODULE_LICENSE("GPL");

static int __init read_disk_init(void)
{
    struct file *fp_read;
    char buf[buf_size];
    int i;
    ktime_t startTime = 0;
```

```
    ktime_t endTime = 0;
    int read_time;
    loff_t pos;
    printk("Start␣read_from_disk␣module...\n");
    fp_read = filp_open("/home/tmp_file", O_RDONLY, 0);
    if (IS_ERR(fp_read)) {
        printk("Failed␣to␣open␣file...\n");
        return -1;
    }

    startTime = ktime_get_real();
    pos = 0;
    for(i = 0; i < read_times; i++) {
        kernel_read(fp_read, buf, buf_size, &pos);
    }
    endTime = ktime_get_real();
    filp_close(fp_read, NULL);
    read_time = (endTime - startTime) / 1000;
    printk(KERN_ALERT "Read␣file␣costs␣%d␣us\n", read_time);
    printk("Reading␣speed␣is␣%d␣M/s\n", buf_size * read_times / read_time);
    return 0;
}

static void __exit read_disk_exit(void)
{
    printk("Exit␣read_from_disk␣module...\n");
}

module_init(read_disk_init);
module_exit(read_disk_exit);
```

6. 编辑 Makefile

```
ifneq ($(KERNELRELEASE),)
    obj-m := read_from_disk.o
else
    KERNELDIR ?= /root/raspberrypi-kernel
    PWD := $(shell pwd)
default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
.PHONY:clean
clean:
    -rm *.mod.c *.o *.order *.symvers *.ko
```

7. 执行 make 编译源码

8. 进行模块加载、查看、卸载

6.3 相关问题解决

编译报错`error: implicit declaration of function 'do_gettimeofday'`

- `do_gettimeofday()`函数在 linux kernel 5.x 之后淘汰
- 使用`ktime_t`直接运算纳秒，除 1000 后即可得到原结果

7 文件管理

7.1 任务 1：为 Ext4 文件系统添加扩展属性（25min）

7.1.1 任务描述

1. 熟悉文件系统扩展属性 EA，查看树莓派文件系统是否支持 EA。
2. 使用 setfattr 设置文件系统的用户扩展属性，并设置文本、八进制数、十六进制数与 base64 编码这四种属性值。
3. 使用 getfattr 获取文件系统的用户扩展属性，并在获取属性之后进行 text、hex 和 base64 这三种编码设置。
4. 分析总结上述实验过程。

7.1.2 审核要求

1. 正确使用 setfattr 设置 EA，正确使用 getfattr 获取 EA。
2. 提交命令行的操作过程截图，以及实验分析总结。

7.1.3 操作指南

1. 安装 libattr

```
# dnf install -y libattr
```

2. 查看文件系统信息

```
# df -Th
# fdisk -l
# tune2fs -l /dev/mmcblklp1 #分别查看上一命令打印的路径
# tune2fs -l /dev/mmcblklp2 #分别查看上一命令打印的路径
```

3. 修改文件属性

- 纯文本属性

```
# cd到目标文件夹
# vi file.txt #创建目标文件
# touch file.txt
# setfattr -n user.name -v SYL file.txt
# setfattr -n user.city -v "NanJing␣SEU" file.txt
# getfattr -d -m file.txt #查看属性
```

- 带有转义字符的属性

```
# setfattr -n user.age -v \012 file.txt
# getfattr -d -m file.txt #查看属性
```

- 十六进制属性

```
# setfattr -n user.hex -v 0x123 file.txt
# 报错，因为0x123为错误十六进制输入
# setfattr -n user.hex -v 0x1234 file.txt
# getfattr -d -m.file.txt #查看属性
```

- Base64 属性

```
# setfattr -n user.base64 -v 0s0123abcd== file.txt
# 报错，因为0s0123abcd==为错误Base64输入
# setfattr -n user.base64 -v 正确的Base64输入 file.txt
# getfattr -d -m.file.txt #查看属性
```

- 属性类型转换为 txt 格式

```
# getfattr -d -e text file.txt
```

- 属性类型转换为十六进制格式

```
# getfattr -d -e hex file.txt
```

- 属性类型转换为 Base64 格式

```
# getfattr -d -e base64 file.txt
```

7.2 任务 2：注册一个自定义的文件系统类型（15min）

7.2.1 任务描述

1. 使用文件系统注册/注销函数，注册一个自定义文件系统类型；
2. 加载模块后，查看系统中是否存在注册的文件系统类型。
3. 加载、卸载模块并查看模块打印信息。

7.2.2 审核要求

1. 正确编写满足功能的源文件，正确编译。
2. 正常加载、卸载内核模块；且内核模块功能满足任务所述。
3. 提交相关源码与运行截图。

7.2.3 操作指南

1. 查看系统中已经注册的文件系统类型

```
# cat /proc/filesystems
```

2. 编辑`register_newfs.c`

```
#include <linux/module.h>
#include <linux/fs.h>

MODULE_LICENSE("GPL");

static struct file_system_type myfs_type = {
    .name  = ">>␣NewFS␣<<",
    .owner = THIS_MODULE,
};
MODULE_ALIAS_FS(">>␣NewFS␣<<");

static int __init register_newfs_init(void)
{
    printk("Start␣register_newfs␣module...");
    return register_filesystem(&myfs_type);
}

static void __exit register_newfs_exit(void)
{
    printk("Exit␣register_newfs␣module...");
    unregister_filesystem(&myfs_type);
}

module_init(register_newfs_init);
module_exit(register_newfs_exit);
```

3. 编辑 Makefile

```
ifneq ($(KERNELRELEASE),)
    obj-m := register_newfs.o
else
    KERNELDIR ?= /root/raspberrypi-kernel
    PWD := $(shell pwd)
default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
.PHONY:clean
clean:
    -rm *.mod.c *.o *.order *.symvers *.ko
```

4. 执行 make 编译源码

5. 进行模块加载、查看、卸载

```
# cat /proc/filesystems
```

7.3 任务 3：在/proc 下创建目录（20min）

7.3.1 任务描述

1. 编写一个模块，在加载模块时，在/proc 目录下创建一个名称为 myproc 的目录；
2. 加载模块后，查看系统中是否在/proc 目录下成功创建 myproc 目录。
3. 加载、卸载模块并查看模块打印信息。

7.3.2 审核要求

1. 正确编写满足功能的源文件，正确编译。
2. 正常加载、卸载内核模块；且内核模块功能满足任务所述。
3. 提交相关源码与运行截图。

7.3.3 操作指南

1. 编辑proc_mkdir.c

```c
#include <linux/module.h>
#include <linux/proc_fs.h>

MODULE_LICENSE("GPL");

static struct proc_dir_entry *myproc_dir;

static int __init myproc_init(void)
{
    int ret = 0;
    printk("Start␣proc_mkdir␣module...");
    myproc_dir = proc_mkdir("myproc",NULL);
    if(myproc_dir == NULL)
        return -ENOMEM;
    return ret;
}

static void __exit myproc_exit(void)
{
    printk("Exit␣proc_mkdir␣module...");
    proc_remove(myproc_dir);
}

module_init(myproc_init);
module_exit(myproc_exit);
```

2. 编辑 Makefile

```
ifneq ($(KERNELRELEASE),)
    obj-m := proc_mkdir.o
else
    KERNELDIR ?= /root/raspberrypi-kernel
    PWD := $(shell pwd)
default:
```

```
7     $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
8 endif
9 .PHONY:clean
10 clean:
11     -rm *.mod.c *.o *.order *.symvers *.ko
```

3. 执行 make 编译源码

4. 进行模块加载、查看、卸载

7.4 任务 4：使用 sysfs 文件系统传递内核模块参数（20min）

7.4.1 任务描述

1. 编写一个模块，该模块有三个参数：一个为字符串型，两个为整型。两个整型中，一个在/sys 下不可见。
2. 加载模块后，使用 echo 向模块传递参数值来改变指定参数的值。
3. 加载、卸载模块并查看模块打印信息。

7.4.2 审核要求

1. 正确编写满足功能的源文件，正确编译。
2. 正常加载、卸载内核模块；且内核模块功能满足任务所述。
3. 提交相关源码与运行截图。

7.4.3 操作指南

1. 编辑sysfs_exam.c

```c
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/stat.h>

MODULE_LICENSE("GPL");

static int a = 0;
static int b = 0;
static char * c = "Hello, World";

module_param(a, int, 0);
MODULE_PARM_DESC(a, "An invisible int under sysfs");
module_param(b, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
MODULE_PARM_DESC(b, "An visible int under sysfs");
module_param(c, charp, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
MODULE_PARM_DESC(c, "An visible string under sysfs");

static int __init sysfs_exam_init(void)
{
    printk("Start sysfs_exam module...");
    printk("a = %d\n", a);
    printk("b = %d\n", b);
    printk("c = '%s'\n", c);
    return 0;
}

static void __exit sysfs_exam_exit(void)
{
    printk("Exit sysfs_exam module...");
    printk("a = %d\n", a);
    printk("b = %d\n", b);
    printk("c = '%s'\n", c);
}
```

```
34
35 module_init(sysfs_exam_init);
36 module_exit(sysfs_exam_exit);
```

2. 编辑 Makefile

```
1  ifneq ($(KERNELRELEASE),)
2      obj-m := sysfs_exam.o
3  else
4      KERNELDIR ?= /root/raspberrypi-kernel
5      PWD := $(shell pwd)
6  default:
7      $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
8  endif
9  .PHONY:clean
10 clean:
11     -rm *.mod.c *.o *.order *.symvers *.ko
```

3. 执行 make 编译源码

4. 进行模块加载、查看、卸载

8 网络管理

8.1 任务 1：编写基于 socket 的 udp 发送接收程序（45min）

8.1.1 任务描述

1. 编写 C 源码，基于 socket 的 udp 发送接收程序，实现客户端与服务端的简单通信。
2. 客户端从命令行输入中读取要发送的内容，服务端接收后实时显示。

8.1.2 审核要求

1. 正确编写满足功能的源文件，正确编译。
2. 提交相关源码与运行截图。

8.1.3 操作指南

1. 编辑 client.c

```c
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>

#define PORT     40000
#define BUF_SIZE    1024

int main(void)
{
    int sock_fd;
    char buffer[BUF_SIZE];
    int size;
    int len;
    int ret;
    struct sockaddr_in server_addr;
    if(-1 == (sock_fd = socket(AF_INET, SOCK_DGRAM, IPPROTO_IP)) ){
        printf("Failed to create a socket!\n");
        return 0;
    }

    //server infomation
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
    bzero(buffer, BUF_SIZE);
    len = sizeof(server_addr);

    //read from stdin and send to server
    while(1){
        printf("Please enter the content to be sent:\n");
        size = read(0, buffer, BUF_SIZE);
        if(size){
            sendto(sock_fd, buffer, size, 0, (struct sockaddr*)&server_addr, len);
```

```
37          bzero(buffer, BUF_SIZE);
38      }
39  }
40  close(sock_fd);
41  return 0;
42  }
```

2. 编辑 server.c

```
1   #include <stdio.h>
2   #include <string.h>
3   #include <sys/types.h>
4   #include <sys/socket.h>
5   #include <arpa/inet.h>
6   #include <unistd.h>
7
8   #define PORT     40000
9   #define BUF_SIZE 1024
10
11  int main(void)
12  {
13      int sock_fd;
14      int len;
15      char buffer[BUF_SIZE];
16      struct sockaddr_in server_addr, client_addr;
17      if(-1 == (sock_fd = socket(AF_INET, SOCK_DGRAM, 0)) )
18      {
19          printf("Failed to create a socket!\n");
20          return 0;
21      }
22      //server information
23      memset(&server_addr, 0, sizeof(server_addr));
24      server_addr.sin_family = AF_INET;
25      server_addr.sin_port = htons(PORT);
26      server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
27      if(-1 == bind(sock_fd, (struct sockaddr*)&server_addr, sizeof(server_addr)))
28      {
29          printf("Failed to bind the socket!\n");
30          return 0;
31      }
32      len = sizeof(client_addr);
33
34      //rec and print
35      while(1)
36      {
37          bzero(buffer, BUF_SIZE);
38          if(-1 != (recvfrom(sock_fd, buffer, BUF_SIZE, 0, (struct sockaddr*)&client_addr, &len)) )
39          {
40              printf("The message received is: %s", buffer);
41          }
42      }
43      return 0;
44  }
```

3. 在同一树莓派中，开启两个终端，一个运行客户端，一个运行服务端；

4. client 中输入发送的消息回车后，server 端即能收到。

8.2 任务 2：使用 tshark 抓包（10min）

8.2.1 任务描述

1. 基于任务 1 的服务端与客户端程序运行时，使用 tshark 抓取该通信数据包

8.2.2 审核要求

1. 正确编写满足功能的源文件，正确编译。
2. 提交相关源码与运行截图。

8.2.3 操作指南

1. 安装 wireshark(可能出现安装失败的情况，尝试安装正确的操作系统版本)

```
# dnf install -y wireshark
```

2. 抓包方法：`tshark [options]` 可通过`tshark -h` 查看具体选项参数
3. 查看网卡信息

```
# ifconfig
```

4. 找到回环地址`lo`
5. 抓包

```
# tshark -i lo -n -f 'udp␣port␣40000' #直接把抓包结果输出到命令行
# tshark -i lo -n -f 'udp␣port␣40000' -T pdml > /root/task2.xml #把抓包结果以指定格式输出到指定文件中
```

6. 按照实验一进行通信
7. 查看抓包结果

8.3 任务 3：使用 setsockopt 发送记录路由选项（25min）

8.3.1 任务描述

1. 基于任务 1 的客户端与服务端，使用 setsockopt 发送一个带 IP 记录路由选项的数据包；
2. 使用 tshark 查看发送的数据包中是否包含了记录路由选项。

8.3.2 审核要求

1. 正确编写满足功能的源文件，正确编译。
2. 提交相关源码与运行截图。

8.3.3 操作指南

1. 编辑 client.c

```c
# #include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>

#define PORT     40000
#define BUF_SIZE    1024

int main(void)
{
    int sock_fd;
    char buffer[BUF_SIZE];
    char rrbuf[28];
    int size;
    int len;
    int ret;

    struct sockaddr_in server_addr;

    if(-1 == (sock_fd = socket(AF_INET, SOCK_DGRAM, IPPROTO_IP)) ){
        printf("Failed to create a socket!\n");
        return 0;
    }

    //server infomation
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
    bzero(buffer, BUF_SIZE);
    len = sizeof(server_addr);

    //read from stdin and send to server
    while(1){
        printf("Please enter the content to be sent:\n");
        size = read(0, buffer, BUF_SIZE);
        if(size){
```

```
39          bzero(rrbuf,sizeof(rrbuf));
40          rrbuf[0] = 0x07;
41          rrbuf[1] = sizeof(rrbuf) - 1;
42          rrbuf[2] = 4;
43          rrbuf[sizeof(rrbuf) - 1] = 0;
44
45          ret = setsockopt(sock_fd,IPPROTO_IP,IP_OPTIONS,(void*)rrbuf,sizeof(rrbuf));
46          if(-1 == ret){
47              printf("setsockopt error!\n");
48              return 0;
49          }
50
51          sendto(sock_fd, buffer, size, 0, (struct sockaddr*)&server_addr, len);
52          bzero(buffer, BUF_SIZE);
53      }
54  }
55  close(sock_fd);
56  return 0;
57 }
```

2. 使用实验一的 server.c

3. 编译并执行代码

```
# ./client
# ./server
# tshark -i lo -n -f 'udp port 40000'-T pdml >./setsockopt.xml
```

4. 查看`setsockopt.xml`内容

9 内核虚拟化

因华为官方对系统版本维护原因，该实验目前暂时无法完成

9.1 任务 1：树莓派 4B 中搭建 openEule 系统的 qemu 虚拟机（50min）

9.1.1 任务描述

1. 在树莓派 4B 中，使用 libvirt+xml 配置文件的方法，搭建 openEuler-20.03-aarch64 系统的 qemu 虚拟机运行环境。
2. 在 windows 主机和树莓派中安装 VNC 工具，完成虚拟机的安装与使用。

9.1.2 审核要求

1. 在树莓派 4B 中正确安装使用 aarch64 架构的 openEuler 虚拟机 qemu 运行环境。
2. 提交关键过程的截图。

9.2 任务 2：在树莓派中搭建和使用 docker（30min）

9.2.1 任务描述

1. 在树莓派的 openEuler 运行环境中，安装使用 docker；完成 docker 容器的新建、启动、守护态运行、终止与删除等操作。
2. 编写 Dockerfile 文件，创建基于 ubuntu 镜像，创建打印“Hello world”的 docker 镜像，并验证其可用性。

9.2.2 审核要求

1. 正确安装、配置与使用 docker。
2. 正确创建功能正常的自定义 docker 镜像。
3. 提交关键流程与截图。